Embedding Logics in Other Logics
Request For Logic (RFL) #4
Robert J. Simmons
December 3, 2009

Existing logical frameworks more-or-less tie one in to a certain type of judgmental methodology. If we embed logics in other logics (like hybrid logics), perhaps we can do more.

1 Embeddings and encodings and adequacy

In the last RFL, I described a bunch of encodings of substructural logics into a variant of the hybrid logical framework that Jason Reed has worked on. But that's not the way that HLF was used in Jason's thesis [JR], which is has a very different quality. While I still don't have a precise and usable way of describing the difference, I want to describe what Jason does in his thesis as *encoding* logics in other logics.

Jason's notes on proof search equivalence in http://jcreed.livejournal.com/1335752.html probably have something to do with a more precise characterization of the difference, but one operational definition that works is this: A logic is encoded in a logical framework by adequately encoding object language propositions as metalanguage terms and then by representing object language judgments as metalanguage propositions. A logic is embedded in a logical framework by giving a function from object language propositions to metalanguage propositions.

Embedding a logic in LLF

In my opinion, the "prettiest" way of embedding polarized intuitionstic logic is into LLF, where neutral object language sequents such as these (lower case letters represent atomic propositions)

$$a^+, B^-, C^- \vdash D^+$$
$$a^+, B^-, C^- \vdash d^-$$

are (respectively) interpreted as the metalanguage sequents such as these:

$$\text{prem } a^+,\ [B^-],\ [C^-];\ ([D^+]\ \text{seq}) \vdash \text{seq}$$
$$\text{prem } a^+,\ [B^-],\ [C^-];\ \text{conc } d^- \vdash \text{seq}$$

Note that a positive proposition is actually interpreted as a *function* from propositions to propositions, so in order to get a "regular old" proposition we have to apply it to another proposition. The interpretation of the right hand side ends up as the only thing in the linear context, and the right-hand side is always the atomic proposition "seq."

The rules for interpreting a function are as follows:

$$[q^-] \Rightarrow (\text{conc } q^- \multimap \text{seq})$$
$$[\uparrow A^+] \Rightarrow \downarrow([A^+]\ \text{seq}) \multimap \text{seq}$$
$$[\top] \Rightarrow \top$$
$$[A^+ \Rightarrow B^+] \Rightarrow [A^+]\ [B^+]$$
$$[A^- \mathbin{\&} B^-] \Rightarrow [A^-]\ \&\ [B^-]$$

$$[q^+] \Rightarrow \lambda N.\ \text{prem } q^+ \Rightarrow N$$
$$[\downarrow A^-] \Rightarrow \lambda N.\ [A^-] \Rightarrow N$$
$$[0] \Rightarrow \lambda N.\ \top$$
$$[1] \Rightarrow \lambda N.\ N$$
$$[A^+ \mathbin{\&} B^+] \Rightarrow \lambda N.\ [A^+]\ ([B^+]\ N)$$
$$[A^+ \oplus B^+] \Rightarrow \lambda N.\ [A^+]\ N \mathbin{\&} [B^+]\ N$$

A few examples:

```
[↓a⁻ ⊃ ↓b⁻ ⊃ c⁻] =>
  (conc a⁻ -o seq) => (conc b⁻ -o seq) => (conc c⁻ -o seq)

[a⁺ ⊃ b⁺ ⊃ ↑c⁺] =>
  prem a⁺ => prem b⁺ => ∀φ. (prem c⁺ => seq) -o seq
```

Embedding a logic in LF

Taking a single step towards the embedding style in [RP], we have the following encoding that works for LF. It uses freshly generated "frame" variables to uniquely tether the interpretation of the object language right hand side to the proposition "seq φ" in the conclusion.

```
[q⁻] => (∀φ. conc q⁻ φ => seq φ)
[↑A⁺] => ↓([A⁺] (seq φ)) => seq φ
[⊤] => ⊤
[A⁺ ⊃ B⁺] => [A⁺] [B⁺]
[A⁻ ∧⁻ B⁻] => [A⁻] ∧ [B⁻]

[q⁺] => λN. prem q⁺ => N
[↓A⁻] => λN. [A⁻] => N
[0] => λN. ⊤
[1] => λN. N
[A⁺ ∧⁺ B⁺] => λN. [A⁺] ([B⁺] N)
[A⁺ ∨ B⁺] => λN. [A⁺] N ∧ [B⁺] N
```

A few examples:

```
[↓a⁻ ⊃ ↓b⁻ ⊃ c⁻] =>
  (∀φ. conc a⁻ φ => seq φ) => (∀φ. conc b⁻ φ => seq φ) => (∀φ. conc c⁻ φ => seq φ)

[a⁺ ⊃ b⁺ ⊃ ↑c⁺] =>
  prem a⁺ => prem b⁺ => ∀φ. (prem c⁺ => seq φ) => seq φ
```

Extension to lax logic

The appendices have two (purported) extensions of the above encoding to lax logic, written in Twelf. One (in Appendix A) uses two judgments, "lax" and "true," which are attached to the right-hand side, but cannot handle an "upshift" embedding a positive proposition in a negative one - this is consistent with the choices made in CLF for other reasons [WCPW]. The second encoding (Appendix B) captures all of polarized lax logic by using a specific judgment "lax" to capture the lax judgment and generating fresh judgment parameters to represent "true."

2 References

[JR] J. Reed, "A hybrid logical framework," Ph.D. dissertation, Carnegie Mellon University, July 2009. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/2009/abstracts/09-155.html

[RP] J. Reed and F. Pfenning, "A constructive approach to the resource semantics of substructural logics."

[WCPW] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker, "A concurrent logical framework: The propositional fragment," Types for Proofs and Programs, pp. 355-377, 2004. [Online]. Available: http://www.springerlink.com/content/fwlrpm102djtlaeb

Appendix A: Embedding a CLF-ish lax logic

Polarity and atomic propositions

```
pol : type.
pos : pol.
neg : pol.

atom : pol -> type.
```

Object language: Polarized intuitionistic logic

Syntax

```
prop : pol -> type.

atm : atom P -> prop P.
↓ : prop neg -> prop pos.
○ : prop pos -> prop neg.

0  : prop pos.
1  : prop pos.
⊤  : prop neg.
⊃ : prop pos -> prop neg -> prop neg.   %infix right 5 ⊃.
∧⁺ : prop pos -> prop pos -> prop pos.  %infix right 7 ∧⁺.
∧⁻ : prop neg -> prop neg -> prop neg.  %infix right 7 ∧⁻.
∨  : prop pos -> prop pos -> prop pos.  %infix right 6 ∨.

%{ We also predefine some atoms to play with. }%

a⁺ : atom pos. qa⁺ = atm a⁺.
a⁻ : atom neg. qa⁻ = atm a⁻.
b⁺ : atom pos. qb⁺ = atm b⁺.
b⁻ : atom neg. qb⁻ = atm b⁻.
c⁺ : atom pos. qc⁺ = atm c⁺.
c⁻ : atom neg. qc⁻ = atm c⁻.
```

Target language, the focused framework FF

Syntax

```
judgment : type.
lax : judgment.
true : judgment.

frame : type.

o : pol -> type.

top : o neg.
∧ : o neg -> o neg -> o neg.        %infix right 6 ∧.
⇒ : o pos -> o neg -> o neg.       %infix right 5 ⇒.
q : atom pos -> o pos.
@ : atom neg -> frame -> o pos.    %infix none 10 @.
seq : frame -> judgment -> o neg.
∀ : (frame -> o neg) -> o neg.
⇓ : o neg -> o pos.
```

Translation

```
trans⁺ : prop pos -> (o neg -> o neg) -> type.
trans⁻ : prop neg -> o neg -> type.
%mode trans⁺ +A -N.
%mode trans⁻ +A -N.

tq+ : trans⁺ (atm Q⁺) ([n] q Q⁺ ⇒ n).
tq- : trans⁻ (atm Q⁻) (∀[ϕ] Q⁻ @ ϕ ⇒ seq ϕ true).
tdn : trans⁺ (↓ A⁻) ([n] ⇓ NA ⇒ n)
       <- trans⁻ A⁻ NA.
tlx : trans⁻ (o A⁺) (∀[ϕ] ⇓(NA (seq ϕ lax)) ⇒ seq ϕ lax)
       <- trans⁺ A⁺ ([n] NA n).

t-0 : trans⁺ 0 ([n] top).
t-1 : trans⁺ 1 ([n] n).
ttp : trans⁻ ⊤ top.
t=> : trans⁻ (A⁺ ⊃ B⁻) (NA NB)
       <- trans⁻ B⁻ NB
       <- trans⁺ A⁺ ([n] NA n).
ta+ : trans⁺ (A⁺ ∧⁺ B⁺) ([n] NA (NB n))
       <- trans⁺ B⁺ ([n] NB n)
       <- trans⁺ A⁺ ([n] NA n).
ta- : trans⁻ (A⁻ ∧⁻ B⁻) (NA ∧ NB)
       <- trans⁻ A⁻ NA
       <- trans⁻ B⁻ NB.
tor : trans⁺ (A⁺ ∨ B⁺) ([n] NA n ∧ NB n)
       <- trans⁺ A⁺ ([n] NA n)
       <- trans⁺ B⁺ ([n] NB n).

%block varframe : block {ϕ : frame}.
%block varjudgment : block {j : judgment}.
%worlds (varframe | varjudgment) (trans⁺ _ _) (trans⁻ _ _).
%unique (trans⁺ +A -C) (trans⁻ +A -B).
%total (A⁺ A⁻) (trans⁺ A⁺ _) (trans⁻ A⁻ _).
```

Examples

Forward-chaining Horn clauses

```
%query 1 * trans⁻ (qa⁺ ⊃ qb⁺ ⊃ ○ qc⁺)
                  (q a⁺ ⇒ q b⁺ ⇒ ∀[ϕ] ⇓(q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻ (qa⁺ ⊃ ○ (qb⁺ ∧⁺ qc⁺))
                  (q a⁺ ⇒ ∀[ϕ] ⇓(q b⁺ ⇒ q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).
```

Backward-chaining Horn clauses

```
%{ Backward-chaining rules shine some light on how the right-hand side of a sequent is
handled. There is no "upshift" in FF, so the only thing that can possibly occur as the
consequent of a stable sequent is the negative atomic proposition `ϕ ◁`. The right-hand
side is turned into a negative proposition. }%

%query 1 * trans⁻ (↓ qa⁻ ⊃ ↓ qb⁻ ⊃ qc⁻)
                  (⇓(∀[φ] a⁻ @ φ ⇒ seq φ true)
                    ⇒ ⇓(∀[ψ] b⁻ @ ψ ⇒ seq ψ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)).
%query 1 * trans⁻ (↓ qa⁻ ∧⁺ ↓ qb⁻ ⊃ qc⁻)
                  (⇓(∀[φ] a⁻ @ φ ⇒ seq φ true)
                    ⇒ ⇓(∀[ψ] b⁻ @ ψ ⇒ seq ψ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)).

%query 1 * trans⁻ (↓(qa⁻ ∧⁻ qb⁻) ⊃ qc⁻)
                  (⇓((∀[φ] a⁻ @ φ ⇒ seq φ true) ∧
                     (∀[ψ] b⁻ @ ψ ⇒ seq ψ true))
                    ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)).

%{
Is this the dual of currying or something?
}%

%query 1 * trans⁻ (↓ qa⁻ ∨ ↓ qb⁻ ⊃ qc⁻)
                  ((⇓(∀[φ] a⁻ @ φ ⇒ seq φ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)) ∧
                   (⇓(∀[ψ] b⁻ @ ψ ⇒ seq ψ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true))).
%query 1 * trans⁻ ((↓ qa⁻ ⊃ qc⁻) ∧⁻ (↓ qb⁻ ⊃ qc⁻))
                  ((⇓(∀[φ] a⁻ @ φ ⇒ seq φ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)) ∧
                   (⇓(∀[ψ] b⁻ @ ψ ⇒ seq ψ true)
                      ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true))).
```

Positive propositions

```
%{
This is cute - the distributivity of products over sums
`A ∧ (B ∨ C) ≡ (A ∧ B) ∨ (A ∧ C)` becomes the distributivity of
exponentiation over products `A ⇒ (B ∧ C) ≡ (A ⇒ B) ∧ (A ⇒ C)` under
the translation.
}%

%query 1 * trans⁻
    (○(qa⁺ ∧⁺ (qb⁺ ∨ qc⁺)))
    (∀[ϕ] ⇓(q a⁺ ⇒ (q b⁺ ⇒ seq ϕ lax) ∧ (q c⁺ ⇒ seq ϕ lax))
       ⇒ seq ϕ lax).

%query 1 * trans⁻
    (○((qa⁺ ∧⁺ qb⁺) ∨ (qa⁺ ∧⁺ qc⁺)))
    (∀[ϕ] ⇓((q a⁺ ⇒ q b⁺ ⇒ seq ϕ lax) ∧ (q a⁺ ⇒ q c⁺ ⇒ seq ϕ lax))
       ⇒ seq ϕ lax).

%query 1 * trans⁻
    (○(qa⁺ ∧⁺ ↓ qb⁻))
    (∀[ϕ] ⇓(q a⁺ ⇒ ⇓(∀[φ] b⁻ @ φ ⇒ seq φ true) ⇒ seq ϕ lax)
       ⇒ seq ϕ lax).
```

Other examples

```
%query 1 * trans⁻
  (qa⁺ ⊃ ↓(qa⁺ ⊃ ○ qb⁺) ⊃ ↓(qb⁺ ⊃ qb⁻) ⊃ ↓(↓ qb⁻ ⊃ qc⁻) ⊃ ○ (↓ qc⁻))
  (q a⁺ ⇒
    ⇓(q a⁺ ⇒ ∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax) ⇒
     ⇓(q b⁺ ⇒ ∀[ϕ] b⁻ @ ϕ ⇒ seq ϕ true) ⇒
      ⇓(⇓(∀[ϕ] b⁻ @ ϕ ⇒ seq ϕ true) ⇒ (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)) ⇒
       (∀[ϕ] ⇓(⇓(∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true) ⇒ seq ϕ lax) ⇒
         seq ϕ lax)).

%query 1 * trans⁻ (qa⁺ ⊃ qb⁻)
                  (q a⁺ ⇒ (∀[ϕ] b⁻ @ ϕ ⇒ seq ϕ true)).

%query 1 * trans⁻ (↓ qa⁻ ⊃ ○ qb⁺)
                  (⇓(∀[ϕ] a⁻ @ ϕ ⇒ seq ϕ true)
                     ⇒ (∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax)).

%query 1 * trans⁻ (○ qb⁺)
                  (∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻ (○(qa⁺ ∧⁺ qb⁺ ∧⁺ qc⁺))
                  (∀[ϕ] ⇓(q a⁺ ⇒ q b⁺ ⇒ q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻ (qa⁻ ∧⁻ qb⁻ ∧⁻ qc⁻)
                  ((∀[ϕ] a⁻ @ ϕ ⇒ seq ϕ true) ∧
                   (∀[ϕ] b⁻ @ ϕ ⇒ seq ϕ true) ∧
                   (∀[ϕ] c⁻ @ ϕ ⇒ seq ϕ true)).

%query 1 * trans⁻ (○(qa⁺ ∨ qb⁺ ∨ qc⁺))
                  (∀[ϕ] ⇓((q a⁺ ⇒ seq ϕ lax) ∧
                            (q b⁺ ⇒ seq ϕ lax) ∧
                              (q c⁺ ⇒ seq ϕ lax))
                    ⇒ seq ϕ lax).
```

Appendix B: Embedding regular-old polarized lax logic

Polarity and atomic propositions

```
pol : type.
pos : pol.
neg : pol.

atom : pol -> type.
```

Object language: Polarized intuitionistic logic

Syntax

```
prop : pol -> type.

atm : atom P -> prop P.
↓ : prop neg -> prop pos.
↑ : prop pos -> prop neg.
○ : prop pos -> prop neg.

0  : prop pos.
1  : prop pos.
⊤  : prop neg.
⊃  : prop pos -> prop neg -> prop neg.   %infix right 5 ⊃.
∧⁺ : prop pos -> prop pos -> prop pos.  %infix right 7 ∧⁺.
∧⁻ : prop neg -> prop neg -> prop neg.  %infix right 7 ∧⁻.
∨  : prop pos -> prop pos -> prop pos.  %infix right 6 ∨.

%{ We also predefine some atoms to play with. }%

a⁺ : atom pos. qa⁺ = atm a⁺.
a⁻ : atom neg. qa⁻ = atm a⁻.
b⁺ : atom pos. qb⁺ = atm b⁺.
b⁻ : atom neg. qb⁻ = atm b⁻.
c⁺ : atom pos. qc⁺ = atm c⁺.
c⁻ : atom neg. qc⁻ = atm c⁻.
```

Target language, the focused framework FF

Syntax

```
judgment : type.
lax : judgment.

frame : type.

o : pol -> type.

top : o neg.
∧ : o neg -> o neg -> o neg.        %infix right 6 ∧.
⇒ : o pos -> o neg -> o neg.       %infix right 5 ⇒.
q : atom pos -> o pos.
@ : atom neg -> frame -> o pos.    %infix none 10 @.
seq : frame -> judgment -> o neg.
∀ : (frame -> o neg) -> o neg.
∀′ : (judgment -> o neg) -> o neg.
⇓ : o neg -> o pos.
```

Translation

```
trans⁺ : prop pos -> (○ neg -> ○ neg) -> type.
trans⁻ : prop neg -> ○ neg -> type.
%mode trans⁺ +A -NA.
%mode trans⁻ +A -N.

tq+ : trans⁺ (atm Q⁺) ([n] q Q⁺ ⇒ n).
tq- : trans⁻ (atm Q⁻) (∀[ϕ] ∀'[j] Q⁻ @ ϕ ⇒ seq ϕ j).
tdn : trans⁺ (↓ A⁻) ([n] ⇓ NA ⇒ n)
       <- trans⁻ A⁻ NA.
tup : trans⁻ (↑ A⁺) (∀[ϕ] ∀'[j] ⇓(NA (seq ϕ j)) ⇒ seq ϕ j)
       <- trans⁺ A⁺ ([n] NA n).
tlx : trans⁻ (○ A⁺) (∀[ϕ] ⇓(NA (seq ϕ lax)) ⇒ seq ϕ lax)
       <- trans⁺ A⁺ ([n] NA n).

t-0 : trans⁺ 0 ([n] top).
t-1 : trans⁺ 1 ([n] n).
ttp : trans⁻ ⊤ top.
t=> : trans⁻ (A⁺ ⊃ B⁻) (NA NB)
       <- trans⁻ B⁻ NB
       <- trans⁺ A⁺ ([n] NA n).
ta+ : trans⁺ (A⁺ ∧⁺ B⁺) ([n] NA (NB n))
       <- trans⁺ B⁺ ([n] NB n)
       <- trans⁺ A⁺ ([n] NA n).
ta- : trans⁻ (A⁻ ∧⁻ B⁻) (NA ∧ NB)
       <- trans⁻ A⁻ NA
       <- trans⁻ B⁻ NB.
tor : trans⁺ (A⁺ ∨ B⁺) ([n] NA n ∧ NB n)
       <- trans⁺ A⁺ ([n] NA n)
       <- trans⁺ B⁺ ([n] NB n).

%block varframe : block {ϕ : frame}.
%block varjudgment : block {j : judgment}.
%worlds (varframe | varjudgment) (trans⁺ _ _) (trans⁻ _ _).
%unique (trans⁺ +A -C) (trans⁻ +A -B).
%total (A⁺ A⁻) (trans⁺ A⁺ _) (trans⁻ A⁻ _).
```

Examples

Forward-chaining Horn clauses

```
%query 1 * trans⁻
  (qa⁺ ⊃ qb⁺ ⊃ ○ qc⁺)
  (q a⁺ ⇒ q b⁺ ⇒ ∀[ϕ] ⇓(q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻
  (qa⁺ ⊃ ○ (qb⁺ ∧⁺ qc⁺))
  (q a⁺ ⇒ ∀[ϕ] ⇓(q b⁺ ⇒ q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻
  (qa⁺ ⊃ qb⁺ ⊃ ↑ qc⁺)
  (q a⁺ ⇒ q b⁺ ⇒ ∀[ϕ] ∀'[j] ⇓(q c⁺ ⇒ seq ϕ j) ⇒ seq ϕ j).

%query 1 * trans⁻
  (qa⁺ ⊃ ↑ (qb⁺ ∧⁺ qc⁺))
  (q a⁺ ⇒ ∀[ϕ] ∀'[j] ⇓(q b⁺ ⇒ q c⁺ ⇒ seq ϕ j) ⇒ seq ϕ j).
```

Backward-chaining Horn clauses

```
%{ Backward-chaining rules shine some light on how the right-hand side of a sequent is
handled. There is no "upshift" in FF, so the only thing that can possibly occur as the
consequent of a stable sequent is the negative atomic proposition `ϕ ◁`. The right-hand
side is turned into a negative proposition. }%

%query 1 * trans⁻ (↓ qa⁻ ⊃ ↓ qb⁻ ⊃ qc⁻)
                  (⇓(∀[φ] ∀′[j] a⁻ @ φ ⇒ seq φ j)
                    ⇒ ⇓(∀[ψ] ∀′[k] b⁻ @ ψ ⇒ seq ψ k)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l)).
%query 1 * trans⁻ (↓ qa⁻ ∧⁺ ↓ qb⁻ ⊃ qc⁻)
                  (⇓(∀[φ] ∀′[j] a⁻ @ φ ⇒ seq φ j)
                    ⇒ ⇓(∀[ψ] ∀′[k] b⁻ @ ψ ⇒ seq ψ k)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l)).

%query 1 * trans⁻ (↓(qa⁻ ∧⁻ qb⁻) ⊃ qc⁻)
                  (⇓((∀[φ] ∀′[j] a⁻ @ φ ⇒ seq φ j) ∧
                     (∀[ψ] ∀′[k] b⁻ @ ψ ⇒ seq ψ k))
                    ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l)).

%{
Is this the dual of currying or something?
}%

%query 1 * trans⁻ (↓ qa⁻ ∨ ↓ qb⁻ ⊃ qc⁻)
                  ((⇓(∀[φ] ∀′[j] a⁻ @ φ ⇒ seq φ j)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l)) ∧
                   (⇓(∀[ψ] ∀′[k] b⁻ @ ψ ⇒ seq ψ k)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l))).
%query 1 * trans⁻ ((↓ qa⁻ ⊃ qc⁻) ∧⁻ (↓ qb⁻ ⊃ qc⁻))
                  ((⇓(∀[φ] ∀′[j] a⁻ @ φ ⇒ seq φ j)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l)) ∧
                   (⇓(∀[ψ] ∀′[k] b⁻ @ ψ ⇒ seq ψ k)
                      ⇒ (∀[ϕ] ∀′[l] c⁻ @ ϕ ⇒ seq ϕ l))).
```

Positive propositions

```
%{
This is cute - the distributivity of products over sums
`A ∧ (B ∨ C) ≡ (A ∧ B) ∨ (A ∧ C)` becomes the distributivity of
exponentiation over products `A ⇒ (B ∧ C) ≡ (A ⇒ B) ∧ (A ⇒ C)` under
the translation.
}%

%query 1 * trans⁻
    (○(qa⁺ ∧⁺ (qb⁺ ∨ qc⁺)))
    (∀[ϕ] ⇓(q a⁺ ⇒ (q b⁺ ⇒ seq ϕ lax) ∧ (q c⁺ ⇒ seq ϕ lax))
       ⇒ seq ϕ lax).

%query 1 * trans⁻
    (○((qa⁺ ∧⁺ qb⁺) ∨ (qa⁺ ∧⁺ qc⁺)))
    (∀[ϕ] ⇓((q a⁺ ⇒ q b⁺ ⇒ seq ϕ lax) ∧ (q a⁺ ⇒ q c⁺ ⇒ seq ϕ lax))
       ⇒ seq ϕ lax).

%query 1 * trans⁻
    (○(qa⁺ ∧⁺ ↓ qb⁻))
    (∀[ϕ] ⇓(q a⁺ ⇒ ⇓(∀[φ] ∀′[j] b⁻ @ φ ⇒ seq φ j) ⇒ seq ϕ lax)
       ⇒ seq ϕ lax).
```

Other examples

```
%query 1 * trans⁻
  (qa⁺ ⊃ ↓(qa⁺ ⊃ ○ qb⁺) ⊃ ↓(qb⁺ ⊃ qb⁻) ⊃ ↓(↓ qb⁻ ⊃ qc⁻) ⊃ ○ (↓ qc⁻))
  (q a⁺ ⇒
    ⇓(q a⁺ ⇒ ∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax) ⇒
     ⇓(q b⁺ ⇒ ∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒
      ⇓(⇓(∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒ (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)) ⇒
       (∀[ϕ] ⇓(⇓(∀[ϕ] ∀′ [j] c⁻ @ ϕ ⇒ seq ϕ j) ⇒ seq ϕ lax) ⇒
         seq ϕ lax)).

%query 1 * trans⁻
  (qa⁺ ⊃ ↓(qa⁺ ⊃ ○ qb⁺) ⊃ ↓(qb⁺ ⊃ qb⁻) ⊃ ↓(↓ qb⁻ ⊃ qc⁻) ⊃ qc⁻)
  (q a⁺ ⇒
    ⇓(q a⁺ ⇒ ∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax) ⇒
     ⇓(q b⁺ ⇒ ∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒
      ⇓(⇓(∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒ (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)) ⇒
       (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)).

%query 1 * trans⁻
  (qa⁺ ⊃ ↓(qa⁺ ⊃ ↑ qb⁺) ⊃ ↓(qb⁺ ⊃ qb⁻) ⊃ ↓(↓ qb⁻ ⊃ qc⁻) ⊃ qc⁻)
  (q a⁺ ⇒
    ⇓(q a⁺ ⇒ ∀[ϕ] ∀′[j] ⇓(q b⁺ ⇒ seq ϕ j) ⇒ seq ϕ j) ⇒
     ⇓(q b⁺ ⇒ ∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒
      ⇓(⇓(∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ⇒ (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)) ⇒
       (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)).

%query 1 * trans⁻ (qa⁺ ⊃ qb⁻)
                  (q a⁺ ⇒ (∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j)).

%query 1 * trans⁻ (↓ qa⁻ ⊃ ○ qb⁺)
                  (⇓(∀[ϕ] ∀′[j] a⁻ @ ϕ ⇒ seq ϕ j)
                     ⇒ (∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax)).

%query 1 * trans⁻ (↓ qa⁻ ⊃ ↑ qb⁺)
                  (⇓(∀[ϕ] ∀′[j] a⁻ @ ϕ ⇒ seq ϕ j)
                     ⇒ (∀[ϕ] ∀′[j] ⇓(q b⁺ ⇒ seq ϕ j) ⇒ seq ϕ j)).

%query 1 * trans⁻ (○ qb⁺)
                  (∀[ϕ] ⇓(q b⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻ (○(qa⁺ ∧⁺ qb⁺ ∧⁺ qc⁺))
                  (∀[ϕ] ⇓(q a⁺ ⇒ q b⁺ ⇒ q c⁺ ⇒ seq ϕ lax) ⇒ seq ϕ lax).

%query 1 * trans⁻ (qa⁻ ∧⁻ qb⁻ ∧⁻ qc⁻)
                  ((∀[ϕ] ∀′[j] a⁻ @ ϕ ⇒ seq ϕ j) ∧
                   (∀[ϕ] ∀′[j] b⁻ @ ϕ ⇒ seq ϕ j) ∧
                   (∀[ϕ] ∀′[j] c⁻ @ ϕ ⇒ seq ϕ j)).

%query 1 * trans⁻ (○(qa⁺ ∨ qb⁺ ∨ qc⁺))
                  (∀[ϕ] ⇓((q a⁺ ⇒ seq ϕ lax) ∧
                            (q b⁺ ⇒ seq ϕ lax) ∧
                              (q c⁺ ⇒ seq ϕ lax))
                    ⇒ seq ϕ lax).
```